TÉCNICO LISBOA

Instituto Superior Técnico

Parallel and Heterogeneous Computing Systems

1º Semester 2021/2022

2º Report
CUDA

Group 7

Nome	Número
Pedro Silva	87103
André Pereira	90016
Filipe Santos	90068

Day of the week: Monday, from 11:30 AM to 2:30 PM.

Professor: Aleksandar Ilic

1 Problem 1

SAXPY is a common linear algebra operation, standing for single-precision A times X plus Y, in which A is a scalar while X and Y are both vectors. This is a simple operation, and one of the functions in the BLAS library.

Its performance was already analyzed in the previous laboratory, with ISPC and ISPC task implementations.

In this laboratory, SAXPY's implementation is a naïve one in CUDA, with a simple kernel where each thread computes each respective index of the result array and the memory transfer is made synchronously before and after the kernel execution. Both these operations are timed, as to get an idea of how the CUDA implementation compares to previous iterations of the program.

Therefore the function that calls the kernel can be divided in memory allocation, memory transfer, kernel call, with (if necessary) CPU synchronization, memory transfer and deletion of allocated space. For all of this memory management, 3 main CUDA functions were used: cudaMemcpy, cudaMalloc and cudaFree; while for the synchronization: cudaDeviceSynchronize.

If ran with the timer wrapping only the kernel, it needs to make use of the latter mentioned function since the kernel calls are asynchronous, and the runtime results end up being a bit faster than even the ISPC implementation, in which the serial version ran in 20ms and the ISPC version ran in 18.9ms for a set of arrays 5 times smaller.

Table 1: SAXPY results without memCpy

Test	$Runtime_{miliseconds}$	$Bandwidth_{GB/s}$
1	27.245	41.020
2	7.731	144.555
3	5.865	190.553

While taking a look at Table 1, it can be expected all subsequent runtime results to be very similar to those of test 3. The first value might seem anomalous, but since the GPU is in idle until the first test run, it is expected to be a bit higher.

It should also be mentioned that, given the fact that the memory bandwidth for this card is 256GB/s, and that the kernel contains only a couple memory accesses, one can also confirm that the bandwidth values don't seem to approach a ludicrous and erroneous value, which does happen when the CPU doesn't wait for the GPU activity to complete.

In the end, it might seem easy to conclude that CUDA outperforms both previous serial and ISPC implementations, but it's not as simple.

Table 2: SAXPY results with memCpy

Test	$Runtime_{miliseconds}$	$Bandwidth_{GB/s}$
1	243.339	4.593
2	262.175	4.263
3	224.438	4.980

When running a program on this GPU, taking into account the memory copy operations to the VRAM, and while ignoring the kernel runtime of approximately 6 seconds, one would expect a 1-way bandwidth of around 16GB/s, given it's connection to a PCIe 3.0 16-lane bus. But instead, the bandwidth only reaches a value of around 5GB/s. This doesn't happen just because of the latency of the PCIe 3.0 connection itself, which is much lower than the runtime ($<$ 1 microseconds), but instead it's an inherent delay associated with the entire API call, making it seem like the PCIe connection is never saturated, and the fact these memory copy functions are synchronous, which can be hidden. To solve this, it is possible to divide both input and output arrays and send these chunks asynchronously, as to make it possible for the kernel to start operating sooner, therefore increasing its performance, but this implementation would be outside of the scope of this first part of the laboratory.

2 Problem 2

2.1 Problem description and Implemented solution

In this problem, we had to find sequential positions of an array that hold the same integer in a parallel implementation, and print the indexes of those positions of the array.

One issue that arises from trying to implement this function in a parallel manner comes when we try to save the indexes of those positions. If we had a vector that held the positions found so far, whenever a thread from the CUDA program tried to append an index, there would most likely be another thread that also wanted to write on that vector. This would create a lot of memory contention, since that to overcome this problem, we would need locks that "serialized" the access to the vector, losing a lot of the potencial efficiency.

In order to explore the capabilities of the GPU, we decided to use the prefix_sum algorithm to resolve this conflict:

- Firstly, we look at the initial array, and store in a different array the comparison (exploring the GPU capabilities) of each array positon with it´s next entry. If both are equal, we store 1, otherwise, we store 0.

Table 3: Input vector

0	7	3	1	1	0	0	2

Table 4: Marked vector

0	0	0	1	0	1	0	0

- We then perform a prefix_sum on the marked vector:

Table 5: Partial vector

0	0	0	0	1	1	2	2

- With the partial vector, we are able to collect all te information needed to, in a parallel manner, fill the output vector with the correct information. By simultaneously analyzing the marked and partial vector in each array entry, we can conclude that when the marked vector has a index that stores 1, we can store on the output vector that index, on the index that is equal to the value stored in the partial vector, and the amount of the index found is simply equal to the last entry of the partial vector.

Table 6: Partial vector

3	5	X	X	X	X	X	X

This way, all the CUDA threads can work in parallel. It is guaranteed that no thread will try to write on the same memory space that another thread is trying to access, thus achieving a parallel implementation to find repeated consecutive entry arrays.

2.2 Performance optimizations

- We performed all the above mentioned operations using the GPU, thus parallelizing all the operations.
- We only launched threads equal to the next power of 2 of threads needed. Meaning, if we only needed to access 13 points of memory to perform operations, we only launched 1 block with 16 threads and ran the necessary operations. If we needed to perform 515 operations, we launched 3 blocks of 256 threads each, and the last block, even though it only needs 3 more threads, it would be ran with the 256 threads, where 253 would remain idle.

Besides this, we also made use of the shared memory capabilites and smart bank usage of the GPU:

- We created shared memory spaces for each block of cuda. By fetching the values only once and having them at the shared memory, we could reduce writing and fetching some values to the global memory, and, instead, if a thread needed to access a memory position that was writen by another thread in the same block, it could just access it directly. In this implementation we did all the sums in the tree for this set of elements (in the block), in one call of the kernel. This achieves a better performance because this program takes longer to access memory than actually doing computations.
- We tried to reduce memory contention of the 32 cuda banks the most we could. For example, (let´s say, for simplicity sake, we only had 8 banks) looking at the first cycle of the upseep, where we add 2 consecutive array positions and store the result in the second:

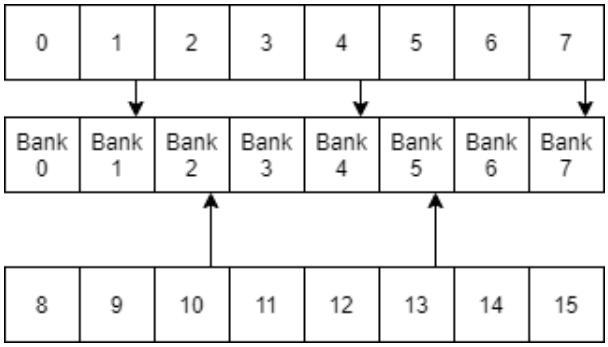

Figure 1: Bank accesses in a array during the first iteration of the upsweep phase. This form of bank usage creates a lot of conflicts.

When thread 1 (which performs the operation on the entry 0 and 1 of the array) and thread 5 (which performs the operation on the entry 8 and 9 of the array) store their result on bank one, they might try to store on the bank at the same time. Since this is not possible, one of the threads will have to wait while the other uses the bank. This becomes a bottleneck factor when a large amount of threads is trying to use the banks all at the same time. By performing padding after each iteration, we can achieve a much better bank usage that leads to much less conflicts (although the conflict are not solved, they still exist, but are lesser):

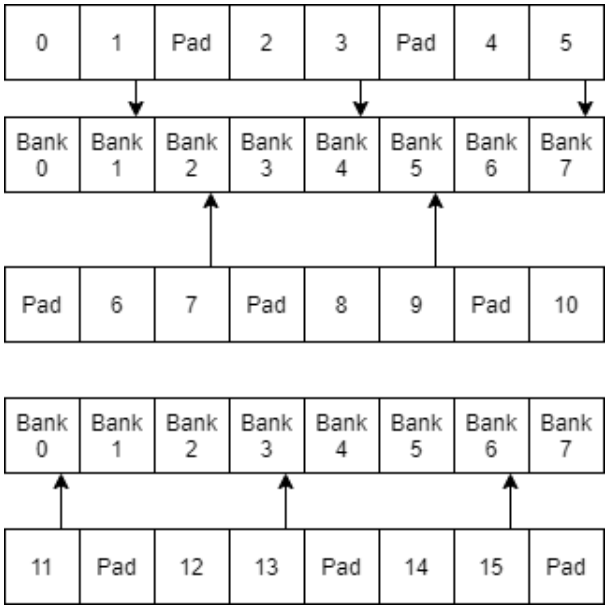

Figure 2: Bank accesses in a array during the first iteration of the upsweep phase after the new distribution. This creates a better shared usage of the banks.

2.3 Performance results

Table 7: Performance Results for scan

Element Count	$Reference_{seconds}$	$Achieved_{seconds}$	Speedup
1000000	2.957	1.673	1,7675
10000000	13.755	8.501	1,618
20000000	27.106	16.704	1,623
40000000	53.822	33.129	1,625

Table 8: Performance Results for find_repeats

Element Count	$Reference_{seconds}$	$Achieved_{seconds}$	Speedup
1000000	3.715	2.639	1,4077
10000000	21.905	16.079	1,36
20000000	32.239	27.529	1,17
40000000	63.599	54.358	1,17

These results show a speed-up relative to a basic implementation of exclusive scan and find repeats. Of course better results could be achieved if the problem was aproached in a different manner, taking into account the low amount of computation in this problem, a higher speed-up could probably be achieved.